Taller de Arquitectura de Software

Refactorización de Monolito – Encuestas Espagueti

- Juan Sebastian Osorio Fierro
- Daniel Steve Fontalvo Matiz
- Leonado Fabio Perez Bermudez
- Juan Camilo Cruz Pardo

FASE 2 – Análisis de Anti-Patrones y Malas Prácticas

Objetivo

Analizar el sistema monolítico e identificar anti-patrones y malas prácticas clasificadas por:

- Seguridad
- Arquitectura
- SOLID
- Clean Code
- Manejo de errores
- Tipado
- Rendimiento

ACTIVIDAD 2.1 – Análisis del Backend (Spring Boot)

Archivo analizado:

EncuestaController.java

1. Seguridad

❌ 1.1 SQL Injection (Concatenación de SQL)

Ubicación

- Método `crear()`
- Método `encuesta()`
- Método `votar()`

Código Problemático

```
String sql = "INSERT INTO encuestas(pregunta, si_count, no_count) VALUES ('" +
pregunta + "', 0, 0)";
String sql = "SELECT id, pregunta, si_count, no_count FROM encuestas WHERE id = "
+ id;
```

🔎 Problema

Las consultas SQL se construyen mediante concatenación de strings.

Esto permite:

- Manipulación del query
- Alteración de la lógica SQL
- Inyección de código malicioso

⚠ Impacto

Vulnerabilidad crítica que compromete la seguridad del sistema.

❌ 1.2 Credenciales Hardcodeadas

```
private String URL = "jdbc:postgresql://db:5432/encuestas";
private String USER = "espagueti";
private String PASS = "espagueti";
```

- 🔎 Problema

1. Credenciales escritas directamente en el código fuente.
2. No se utilizan variables de entorno.
3. No se emplea application.properties o application.yml.

⚠ Impacto

- Riesgo de exposición de credenciales.
- Mala práctica DevOps.
- No es portable a producción.
- No cumple principios de configuración externa.

🏗 2. ARQUITECTURA

❌ 2.1 Ausencia de Arquitectura en Capas

El EncuestaController realiza:

1. Conexión a base de datos
2. Lógica de negocio
3. Validaciones
4. Construcción de SQL
5. Manejo de errores

- 🔎 Problema

No existen capas separadas como:

1. Controller
2. Service
3. Repository
4. DTO
5. Configuración de DataSource
6. Todo se encuentra en una sola clase.

⚠ Impacto

- Alta acoplación
- Difícil mantenimiento
- Baja escalabilidad
- Difícil testeo unitario
- Violación del principio de separación de responsabilidades

🧱 3. PRINCIPIOS SOLID

❌ 3.1 Violación del SRP (Single Responsibility Principle)

El controlador:

- Maneja lógica de negocio
- Gestiona conexión a BD
- Ejecuta SQL
- Valida datos
- Maneja errores
- Construye respuestas

🔎 Problema

Un controlador debería:

1. Recibir la request
2. Delegar la lógica al Service
3. Retornar la respuesta

- Actualmente hace múltiples responsabilidades.

⚠ Impacto

- Código difícil de mantener
- Baja cohesión
- Alto acoplamiento
- Difícil evolución futura

4. CLEAN CODE

❌ 4.1 Código Duplicado

Ejemplo de validaciones repetidas:

```
if (body == null || body.get("pregunta") == null ||
body.get("pregunta").toString().trim().length() < 3)
```

Este patrón se repite en múltiples métodos.

🎾 Problema

1. Validaciones repetidas.
2. No existe método reutilizable.
3. No se centraliza la lógica.

⚠ Impacto

- Mayor probabilidad de errores.
- Código más largo de lo necesario.
- Difícil mantenimiento.

5. MANEJO DE ERRORES

❌ 5.1 Uso de printStackTrace() y return null

```
catch (Exception e) {
    e.printStackTrace();
}
return null;
```

🎾 Problema

1. Solo imprime el error en consola.
2. Retorna null.
3. No se devuelve un código HTTP adecuado.
4. No se utiliza @ExceptionHandler.

⚠ Impacto

- Respuestas inconsistentes.
- Mala práctica REST.
- Dificulta monitoreo y trazabilidad.
- Riesgo de NullPointerException en cliente.

🧾 6. TIPADO DEFICIENTE

❌ 6.1 Uso de Map sin Genéricos ni DTOs

```
public Map crear(@RequestBody Map body)
```

🔎 Problema

1. Uso de Map sin tipado genérico.
2. No existen DTOs.
3. No hay validación estructurada.
4. No hay clases de dominio.

⚠ Impacto

- Pérdida de tipado fuerte.
- Mayor riesgo de errores en tiempo de ejecución.
- Código menos mantenible.

🚀 7. RENDIMIENTO

❌ 7.1 Conexión BD Recreada en Cada Llamada

```
private JdbcTemplate jdbc() {
    DriverManagerDataSource ds = new DriverManagerDataSource();
```

- Cada vez que se ejecuta un endpoint:

1. Se crea un nuevo DataSource.
2. Se crea un nuevo JdbcTemplate.

3. Se genera una nueva conexión.

🔎 Problema

No se utiliza:

1. Pool de conexiones
2. Configuración centralizada de DataSource

⚠ Impacto

- Ineficiencia
- Bajo rendimiento bajo carga
- No escalable

💻 ACTIVIDAD 2.2 – Análisis del Frontend (Angular)

- Archivos analizados:

```
crear.component.ts

encuesta.component.ts

respuestas.component.ts
```

❌ 1. URL Hardcodeada

this.http.post("http://localhost:8081/crear", ...)

🔎 Problema

- No se usa environment.ts.
- No hay configuración por entorno.

⚠ Impacto

1. No portable a producción.
2. Mala práctica de configuración.

❌ 2. Uso Directo de HttpClient en el Component

constructor(private http: HttpClient)

🔎 Problema

1. No existe una capa Service intermedia.

2. Se viola el patrón de arquitectura Angular recomendado.

⚠ Impacto

1. Violación del patrón Service.
2. Lógica mezclada con presentación.
3. Dificulta pruebas unitarias.

❌ 3. Manipulación Directa del DOM

document.getElementById(...)

🎾 Problema

Angular debe utilizar:

1. Data Binding
2. Directivas
3. Templates reactivos
4. Manipular el DOM directamente es un anti-patrón.

⚠ Impacto

- Código menos mantenible.
- Rompe el enfoque declarativo de Angular.

❌ 4. Uso Excesivo de any

(r: any)

🎾 Problema

1. No hay interfaces tipadas.
2. No hay modelos de datos.
3. Se pierde el beneficio de TypeScript.

⚠ Impacto

- Riesgo de errores.
- Pérdida de autocompletado fuerte.
- Código menos robusto.

❌ 5. Polling Manual con setInterval

Uso de:

```
setInterval(...)


En lugar de:

- Observables
- RxJS
- AsyncPipe
```

⚠ Impacto

- Código menos reactivo.
- No aprovecha el paradigma Angular.
- Posible consumo innecesario de recursos.

📌 CONCLUSIÓN TÉCNICA – FASE 2

El sistema presenta múltiples anti-patrones y malas prácticas:

1. Vulnerabilidad a SQL Injection
2. Credenciales hardcodeadas
3. Violación del principio SRP
4. Ausencia de arquitectura en capas
5. Manejo deficiente de errores
6. Bajo rendimiento por mala gestión de conexiones
7. Uso incorrecto de patrones en Angular

Se trata de un monolito funcional pero mal estructurado, lo que justifica una refactorización arquitectónica hacia:

- Arquitectura en capas (Controller → Service → Repository)
- Uso de DTOs
- Configuración externa
- Uso de Prepared Statements
- Implementación de pool de conexiones
- Separación adecuada en frontend (Service layer)